The version of record of this manuscript has been published and is available in Communications of the ACM, Vol. 60 No. 11, Pages 26-28. 10.1145/3144174

Keeping the Machinery in Computing Education

Richard Connor, University of Strathclyde, Scotland, Richard.Connor@strath.ac.uk

Quintin Cutts*, University of Glasgow, Scotland, Quintin.Cutts@glasgow.ac.uk

Judy Robertson, University of Edinburgh, Scotland, Judy.Robertson@ed.ac.uk

**Corresponding Author*

We don't think there can be "computer science" without a computer. Some efforts at deep thinking about computing education seem to sidestep the fact that there is technology at the core of this subject, and an important technology at that. Computer science practitioners are concerned with making and using these powerful, general-purpose engines. To achieve this, computational thinking is essential; however so is a deep understanding of machines and languages, and how these are used to create artefacts. Efforts to make computer science entirely about "computational thinking", in the absence of "computers", are mistaken, in our opinion.

As academics we were invited to help develop a new curriculum for computer science in Scottish schools, covering ages 3-15. We proposed a single coherent discipline of computer science with foundations running from this early start, similar to disciplines such as maths. Pupils take time to develop deep principles in those disciplines, and with appropriate support the majority of pupils make good progress. From our background in CS education research, we saw an opportunity for *all* children to learn useful foundations. Nobody knows exactly the right CS curriculum for the average five-year old, as we've not taught them CS before, but we are unconvinced of the coherence of many current curricula: an underlying intellectual and developmental framework seems to be missing, and such a framework is our principal offering to the curriculum.

We understand both the desperate calls from industry to meet the labour market demands of the digital economy, and the extraordinary environment that will be our children's, with ever more blurring of digital and human worlds. Hence, we wanted a curriculum that properly grounds their understanding of that non-human world and gives every child the opportunity, should they wish, of a future career in our area. Our school systems have these aspirations in teaching about the natural world – why not the digital world also?

The version of record of this manuscript has been published and is available in Communications of the ACM, Vol. 60 No. 11, Pages 26-28. 10.1145/3144174

In March 2017, the new curriculum was formally adopted at government level, and its delivery has started. A teachers' guide is here: www.teachcs.scot and we encourage interested readers to look at the full guide there.

All curriculum design requires compromise. We have balanced: our initial vision of a curriculum that captures the essence of computation at the heart of the digital revolution; the practical realisation that only a small amount of resource is available for teachers' professional development; the requirement to re-use a varied body of existing early-years computing educational material; and the desire from government to direct computing education down a narrow agenda to fill a perceived skills shortage.

Nonetheless, we have kept in view throughout our overarching framework consisting of three main points that we think is the real contribution of the curriculum, and the three points are the focus of this Viewpoint. In the following sections we show: the essence of our proposed three-point underpinning, developing three essential strands of learning, and the way these have been eventually interpreted in the adopted curriculum. The success, or otherwise, of our core ideas remains to be seen!

Computational Foundations

We aimed to identify a core framework for the discipline that is equally relevant to a child, a university student or a software engineer.

The essence of computation is clear: the Church-Turing thesis. Some kind of computational mechanism --- whether the Scratch programming environment, a Turing Machine, or the Lambda Calculus --- can be used to model any tail-recursive numeric function… and therefore anything which can be computed… and furthermore all such mechanisms are somehow equivalent.

To be of interest, such mechanisms should be restricted to those which can perform some kind of *modelling* function over another domain or world. That is, they can be set up in such a way that their operation, when viewed in the context of the other domain, can be seen as simulating some aspect of that domain. Hence a programming language can be used to model an aspect of the real world; a processor can be set up with appropriate machine code to model a computation expressed in a programming language; a lambda calculus expression, under the application of reduction, can provide the result of some recursive function.

The version of record of this manuscript has been published and is available in Communications of the ACM, Vol. 60 No. 11, Pages 26-28. 10.1145/3144174

A deep understanding of computer science requires the following three aspects, our three-point framework, which can be neatly separated as the understanding of:

1. Domains that can be modelled by computational mechanisms,
2. The computational mechanisms themselves, and
3. How to use the computational mechanisms to model aspects of the domains.

It is our belief that a computer scientist is habitually and implicitly aware of these, and indeed is expert at quickly assimilating new instances of them. We believe this is a core skill with many applications to a modern process- and information-driven world.

Computational Thinking, as well as the learning delivered via the Unplugged approach, are, we believe, largely captured within the first aspect. The skill of programming, as taught even at university level, is mostly within the third. The second all-important aspect seems to be often neglected, at least until the later stages of a computing degree. It has long been a wry observation of the authors that, while “programming” is taught right from the start of university computing courses, more “advanced” topics such as programming language syntax and semantics are typically taught much later on. This begs the question: how can one learn to program in the absence of such knowledge? Research shows that concentrating on explaining how programs work, rather than writing them, helps students early on to learn programming. Could it be that we normally teach “by example” only, rather than ever properly defining the domain in which the modelling is performed, or even the domain being modelled?

Our Curriculum

The resulting curriculum is formally structured around these three aspects. Here we outline how they are presented to non-computer scientists – see the detail at www.teachcs.scot. The vocabulary and concepts used are accessible to those who need to read them; the difficulty of this should not be under-estimated, it is hard for an academic computer scientist to communicate with a teacher of early years computing.

Each of our three main aspects persists through the five defined levels of the curriculum, from age three to fifteen; the text here is mostly aimed at teachers of the lower levels.

The version of record of this manuscript has been published and is available in Communications of the ACM, Vol. 60 No. 11, Pages 26-28. 10.1145/3144174

1: Understanding the world through computational thinking

The first aspect looks at the underlying theory in the academic discipline of Computing Science. Theoretical concepts of Computing Science include the characteristics of information processes, identifying information, classifying and seeing patterns.

This aspect is about understanding the nature and characteristics of **processes** and **information**. These can be taught through Unplugged activities (fun active learning tasks related to Computing Science topics but carried out without a computer) and with structured discussions with learners. There is a focus on recognising computational thinking when it is applied in the real world such as in school rules, finding the shortest or fastest route between school and home, or the way objects are stored in collections.

Learners will be able to identify steps and patterns in a **process**, for example seeing repeated steps in a dance or lines of a song. In later stages, learners will begin to reason about properties of processes, for example considering whether tasks could be carried out at the same time, whether the output of a process is predictable, and how to compare the efficiency of two processes.

Learners will identify **information**, classify it and see patterns. For example, learners might classify and group objects where there is a clear distinction between types or where objects might belong to more than one category.

2: Understanding and analysing computing technology

This aspect aims to give learners insight into the hidden mechanisms of computers and the programs that run on them. It explores the different kinds of language, graphical and textual, used to represent processes and information. Some of these representations are used by people and others by machines, for example, a verbal description, a sequence of blocks in a visual programming language such as Scratch, or as a series of 1s and 0s in binary.

In this aspect, learners will learn how to ‘read’ program code (before writing it in the next aspect) and describe its behaviour in terms of the **processes** they have learned about in the first aspect, processes that will be carried out by the underlying machinery when the program runs. For example, learners could read a section of code and predict what will happen when it runs or if lines of code change order. Learners will learn and explore different representations of **information** and how these are stored and manipulated in the computing system under study.

The version of record of this manuscript has been published and is available in Communications of the ACM, Vol. 60 No. 11, Pages 26-28. 10.1145/3144174

3: Designing, building and testing computing solutions

The third aspect is about taking the concepts and understanding from the first two aspects and applying them. Learners will create solutions, perhaps by designing, building and testing solutions on a computer or by writing a computational process down on paper. In doing so, they will learn about modelling process and information from the real world in programs, and what makes a good model to represent or solve a particular problem.

Learners will create representations of information. For example, learners could make lists, tables, family trees, Venn diagrams and data models to capture key information from the problems they are working on.

Learners will use their skills in language to create descriptions of processes that can be used by other people. For example, a computer program is a great way to describe a process.

Learners will understand how to read, write and translate between different representations such as between English statements, planning representations and actual computer code. For example, developing skills in writing code could be scaffolded by studying worked examples or by giving learners jumbled lines of code and asking them to put the lines into an appropriate order.

Although solutions can be created in a many ways, it is expected that all learners will experience creating solutions on computers. This shows learners that computers implement exactly what they, the learners, have written and not what they intended, as well as giving them practice in debugging.

Reflections

We have presented a curriculum which explicitly connects computational thinking with the more mechanical aspects of computing, with particular concentration on the explicit modelling of computational domains by computational mechanism. Not everyone needs to become a software engineer or computer scientist; the curriculum provides valuable learning at all levels, including the essential foundations for those who wish to study the subject further. While our curriculum is informed by previous educational computing research, we emphasise quite different learning outcomes via our three-point framework.